god-stack-generative-agentic-ai

Welcome to the **FullStack AI & LLM Engineering** Repo. `Python`, `Git`, `Docker`, `Pydantic`, `LLMs`, `Agents`, `RAG`, `LangChain`, `LangGraph`, and `Multi-Modal AI` from the ground up.

This is not just another theory repo. By the end,we would **code, deploy, and scale real-world AI applications** that use the same techniques powering **ChatGPT, Gemini, and Claude**.

Real-World Projects We Built

- Tokenizer from scratch.
- Local `Ollama` + `FastAPI` AI app.
- `Python` CLI-based coding assistant.
- Document RAG pipeline with `LangChain` & Vector DB.
- Queue-based scalable RAG system with `Redis` & `FastAPI`.
- AI conversational voice agent (STT + GPT + TTS).
- Graph memory agent with `Neo4j`.
- MCP-powered AI server.

Chapters

Tech Stack

Foundations

- `Python` programming from scratch — syntax, data types, OOP, and advanced features.
- `Git` & `GitHub` essentials — branching, merging, collaboration, and professional workflows.
- `Docker` — containerization, images, volumes, and deploying applications like a pro.
- `Pydantic` — type-safe, structured data handling for modern Python apps.

AI Fundamentals

- What are LLMs and how `GPT` works under the hood.
- Tokenization, embeddings, attention, and transformers explained simply.
- Understanding multi-head attention, positional encodings, and the "Attention is All You Need" paper.

Prompt Engineering

- Master prompting strategies: zero-shot, one-shot, few-shot, chain-of-thought, persona-based prompts.
- Using `Alpaca`, `ChatML`, and `LLaMA-2` formats.
- Designing prompts for structured outputs with `Pydantic`.

Running & Using LLMs

- Setting up `OpenAI` & `Gemini` APIs with `Python`.
- Running models locally with `Ollama` + `Docker`.
- Using `Hugging Face` models and INSTRUCT-tuned models.
- Connecting LLMs to `FastAPI` endpoints.

Agents & RAG Systems

- Build our first AI Agent from scratch.
- CLI-based coding agents with `Claude`.
- The complete **RAG pipeline** — indexing, retrieval, and answering.
- `LangChain`: document loaders, splitters, retrievers, and vector stores.
- Advanced RAG with `Redis/Valkey Queues` for async processing.
- Scaling RAG with workers and `FastAPI`.

LangGraph & Memory

- Introduction to `LangGraph` — state, nodes, edges, and graph-based AI.
- Adding checkpointing with `MongoDB`.
- Memory systems: short-term, long-term, episodic, semantic memory.
- Implementing memory layers with `Mem0` and Vector DB.
- Graph memory with `Neo4j` and `Cypher` queries.

Conversational & Multi-Modal AI

- Build voice-based conversational agents.
- Integrate speech-to-text (STT) and text-to-speech (TTS).
- Code our own **AI voice assistant for coding (Cursor IDE clone)**.
- Multi-modal LLMs: process images and text together.

Model Context Protocol (MCP)

- What is MCP and why it matters for AI apps.
- MCP transports: STDIO and SSE.
- Coding an MCP server with `Python`.

Table of Contents

1 Agentic AI Developer Environment Setup

This documentation covers the necessary steps and recommended tools for setting up the developer environment for the Agentic AI with Python course. This is the first video of the course.

I. Overview of Required Tools

The tools needed for the development environment setup are free and very easy to set up. It is likely that these tools are already available on the user's machine.

II. Mandatory Environment Setup

1. Integrated Development Environment (IDE)

An IDE is needed for coding.

- **Recommended Tool:** The presenter will be using **Visual Studio Code (VS Code)**.
- **Recommendation Level:** It is highly, highly, highly recommended to use VS Code to easily follow along in the tutorial/video.
- **Installation:** VS Code is absolutely free to download. Users must install VS Code as an IDE. Users can choose platforms, such as Windows or Linux.
- **Flexibility:** Users are free to use any other IDE if preferred.

2. Python Installation

Python is mandatory for this course. The entire course is based on Python.

- **Installation Requirement:** Please make sure that Python is installed on the machine.
- **Installation Steps:** Installation is very simple; users can click the download button, click "next, next, next," and follow the on-screen instructions.
- **Version Check Example:** The presenter showed the installed version on their terminal:

```
Python version 3.1, 3.2 installed
```

3. Required Python Knowledge

The assumption is that the user already knows the basics of Python. Users do not need to be a "pro" in Python.

Required basic knowledge includes:

- Knowing what variables are.
- Knowing how to define functions.
- Knowing how to make variables.
- Understanding simple mathematical operations.
- Understanding simple classes.

This level of information is considered good enough.

4. AI Service Accounts

Because this is an agentic AI course, it will utilize Large Language Models (LLMs).

- **Required LLM Awareness:** Users should be aware of **ChatGPT**.
- **Account Requirement:** Users must have an account on **OpenAI**.

- **LLMs Used in Course:** The course will use **OpenAI** and **Gemini**, along with other LLMs discussed in upcoming videos.
- **Setup Note:** There will be a dedicated video on how to set up an OpenAI account, how to sign up, how to add credits to OpenAI, or how to use Gemini.

III. Optional VS Code Customization Setup

This section details the specific setup, themes, and extensions used by the presenter. This video is absolutely optional and can be skipped.

1. VS Code Themes

The presenter uses specific themes for the interface and icons.

- **Overall Theme:** The theme used throughout the course is **Aumiraj**.
  - The specific theme being currently used is the **AO Mirage dark border theme**.
- **Icon Theme:** The theme used for icons is **Material Icon Theme**.

2. VS Code Extensions

These are the extensions currently installed in the presenter's VS Code environment.

Recommended Extensions for Python Developers

These extensions are highly recommended to install:

1. **Pylance:** This is a main extension used a lot by the presenter. It is developed by Microsoft.
2. **Python Language for VS Code:** This is an extension that is highly recommended to install.
3. **Python Debugger:** This extension is good to have. It is also developed by Microsoft. Although the presenter does not use it a lot, it is highly recommended to install.
4. **Prettier:** This extension is useful for formatting code and general code cleanliness.

Other Extensions Used by the Presenter

- **TypeScript Extensions:** Installed because the presenter works on TypeScript frequently.
- **.NET Extensions:** Some are installed.
- **Container Tools:** Installed for working with Docker. These tools include:
  - `dev containers`.
  - `Docker`.
  - `Docker DS`.

These listed items are the major extensions needed as a Python developer.

2 Introduction to Python Programming and Tools

I. Course Introduction and Philosophy

Instructor Background and Teaching Style

The instructor's name is **Hitesh**. He did engineering in electronics and communications, not computer science, demonstrating that anyone can learn to code. He has worked in the Python ecosystem, cybersecurity, iOS development, web development, databases, and JavaScript. He currently runs two startups, one of which serves around 22 million users, emphasizing his ability to scale things and write great quality software.

His specialty is turning the toughest topic into the easiest one. He reads a lot of books about teaching mechanisms, how to deliver engaging content, and tech topics like Python and JavaScript.

Two styles of learning explored:

1. **First Style (First Principle Learning):** Getting into the details and exploring every bit to see that there is "no magic" in frameworks, and recreating features.
2. **Investigative Learning (Exploration/Hunting/Experimenting):** Investigating things as they are written. This entire course will focus on this **investigative style of learning**, questioning every output and every line of code.

The teaching style is **laid back** and not fast-paced, allowing the brain time to process and permanently save the information into secondary storage. It is important not to rush the process (e.g., watching at 1.5x or 2x speed) because the brain needs time to process this.

Course Tools for Instruction

The instructor uses a mix of tools for explanation and coding.

Tool/Item	Purpose/Description	Citation
VS Code	The chosen code editor.	
Chai Theme	A dark color theme created by the instructor. It makes the code look beautiful. Search for `chai theme` in extensions.	
Python Extensions (e.g., Pylance)	Extensions that provide color codes and type hinting, helping users write code without typing everything manually.	
Warp	A preferred terminal software, available on Mac, Windows, and Linux.	
Eraser	A tool used to teach by drawing diagrams (like a whiteboard/blackboard), linking boxes and moving them.	
tldraw & Excalidraw	Other diagramming tools used extensively by the instructor.	

The instructor may switch visibility modes, sometimes appearing on screen for engagement, and other times switching entirely to the screen view for 100% focus.

II. Programming Fundamentals: Converting Instructions to Code

What is Programming?

Programming is about **giving instruction to a computer**. Crucially, these instructions must be ones that the computer understands. Computers cannot think on their own; they must be provided with exact instructions. Even AI relies on repeating steps seen extensively on the internet and is essentially fancy word completion.

Coding requires an **extensive amount of effort over the years**. It is not a walk in the park, but it is doable, especially with languages like Python or JavaScript. While simple code can be written within a few months, truly mastering programming requires time. Writing code is the easy part; the tough part is **thinking about it** and having the thought process of a programmer.

Programming involves three major components, demonstrated by the "making chai" analogy:

1. **Gathering the stuff (Collection of Data)**: E.g., gathering water, milk, tea leaves, sugar, and utensils.
2. **Checking conditions**: E.g., having enough water, or clean cups.
3. **Thorough steps (Precise instructions)**: The exact process of making the tea.

Example: Steps for Making Tea

The following steps outline the process of making tea:

1. Check if the kettle has water.
2. Plug in the kettle.
3. Boil water.
4. Get a clean cup or cups.
5. Add T leaves and sugar to the cup (or boiling water).
6. Pour boiled water into the cup (and milk, if preferred).
7. Stir and serve.

III. Tools and Setup: Installation and Running Python

Installing Python and VS Code

Software is generally very compatible across Mac and Windows.

1. **Download Source:** Python can be downloaded from `python.org`. It usually detects the system and provides the appropriate version; otherwise, the user can select their specific version (e.g., Windows, Mac, Linux). It is recommended to choose the latest version.
2. **Installation (Windows Specific):**
   - Use the default installer (which is "next next I agree").
   - When installing, the first thing is to **add a path** (so it doesn't have to be manually added) and use **admin privileges**.
   - Sometimes, Windows may show warnings about the path being too long; the user can disable the length check and allow it.
3. **Download Code Editor:** VS Code is the preferred editor and can be downloaded for Windows, Mac, or Linux. Installation is simple.

Running Python and Verification

Python programs can be written in two ways: in the shell (terminal) or into a file with the special extension `.PY` so that the code can be saved and reused. The second method (using files) is preferred.

Verifying Installation (using Terminal/Shell):

1. **Open Terminal:** Open any preferred browser or terminal (like Warp).
2. **Command:** Type the command to check the version.

   - **Mac:** `python3 --version`. (Mac defaults to Python 2, so `python3` must be used).
   - **Windows:** `Python --version`.
3. **Verification:** If the command provides an output (e.g., `Python 3.13.2`), installation is successful.

Running Python in the Shell (Interactive Mode): To run the Python shell mode, type `python3` (Mac) or `python` (Windows) and hit enter.

Example commands in the shell:

```
>>> 2 + 2
4
>>> import sys
>>> print(sys.version)
(Outputs the system version)
>>> exit()
```

The shell allows immediate execution, but the code is lost afterward. `exit()` or `exit` is used to leave the shell.

Running Python from a File (VS Code):

1. **Set up Workspace:** Drag and drop a folder (e.g., `Test Win` or `Python udemy`) into VS Code.
2. **Create File Structure:** Create a folder (e.g., `01basics`) and a new file inside it (e.g., `Python Test document.py` or `test Python py`). The extension **.PY is crucial** as it tells the compiler it must be run by the Python interpreter.
3. **Write Code:** Write a basic, decent Python code snippet.
   - `import sys` (imports system libraries).
   - `print(sys.version)` (prints the system version).
4. **Open Terminal in VS Code:** Use the shortcut (Command + Tilde or Control + Tilde) or the menu.
5. **Run Command:** Use `python` (or `python3` on Mac) followed by the path to the file.

Example execution command:

```
$ Python 3 01basics/testPython.py
(Outputs the expected system version)
```

IV. Python Basics: Syntax and Concepts

Functions, Methods, and Indentation

In programming, instructions are wrapped in a container known as a **function** or a **box**. Functions are sometimes used interchangeably with the term **methods**. Anything followed by parentheses `()` is a function.

The basic syntax to create a function/box:

```
def makechai(): # definition of the function
    # instructions here
```

A function needs to be **called** to execute the instructions inside it.

```
makechai() # calling the function
```

Indentation: Python relies on **indentation**.

- It is recommended to use **four spaces** for indentation.
- Using a tab key is usually not recommended, even though tabs often internally provide four spaces.
- Lack of indentation will cause the code not to work.

Conditional Logic (Example using Non-Pythonic Code): The code can be read almost like English.

```
def makechai(): # Defines the "box"
    if not kettle has water: # Checks if the condition is false
        fill cattle # Executes if condition is true (no water)

    plug in the cattle # Another function/instruction
    boil water # Another function/instruction

    if not is cup clean?:
        wash the cup
        # or pick another cup

    add to cup(T leaves)
    add to cup(sugar)

    pouring(to: pour, what: boiled water)
    stir(cup)

    serve, chai

makechai() # Calls the function to start the process
```

This example shows that if you can read English (e.g., "if not kettle has water"), you can read Python code. This non-accurate code demonstrates functions, conditions, methods, and syntax.

Core Object-Oriented Concepts (O-O)

A Python program can be a mix of classes, methods, objects, and properties.

Concept	Definition/Analogy	Example	Citation
Object	A "thing" in the world of programming.	`cup`, `kettle`, `chai`	
Property	Characteristics or attributes of an object.	`cup color`, `chai sweetness`	

Concept	Definition/Analogy	Example	Citation
Method / Function	The actionable step or simple **actions** (performing something).	`stirring`, `pouring`, `drink`, `sip`, `add sugar`	
Class	A whole big box or "factory" that contains smaller units (functions/methods/properties).	`Chai`, `ChaiShop`	

Advanced Code Example (Class Structure)

This example demonstrates a `class` and the use of the initialization method (`__init__`). The extension `.py` is used for Python files.

Note: This is an experience part; the user does not have to write this code.

```
class Chai:
    # This is the initialization method; it runs as soon as the factory is opened
    # It sets up the blueprint (e.g., default sweetness)
    def __init__(self, sweetness, milk_level):
        self.sweetness = sweetness
        self.milk_level = milk_level
        # 'self' is a keyword that will be explored in depth

    # Factory/Method 1: Sipping
    def sip(self):
        # 'self' must be included when defining anything inside the class
        print("sipping chai")

    # Factory/Method 2: Adding sugar
    def add_sugar(self, amount):
        print("added the sugar")

# How to use the class:
# 1. Create an object ('my_chai') from the blueprint ('Chai')
# 2. This initializes the class (kicks off __init__) and requires required
parameters
my_chai = Chai(3, 50) # Sweetness=3, Milk level=50

# 3. The object can now run any methods defined in the class
my_chai.add_sugar(5) # Call the method to add sugar
```

The `__init__` method must be provided whenever a class is opened/initialized. The object (`my_chai`) uses a dot (`.`) to access methods like `add_sugar` or `sip`.

V. Organizing Python Code

Organizing Python code is crucial, although there are many opinions on the right way to structure it.

Structure Definitions

Name	Definition/Characteristics	Example	Citation
Module	**Every single file** that is a normal Python file.	`run.py`, `chai.py`	
Package	**Any folder** which contains the file `__init__.py`. The `__init__.py` file is empty; only its name matters.	`utils` (if it contains `__init__.py`)	
Other Folders	Folders that do not contain `__init__.py`.	`processing` (if it lacks `__init__.py`)	

A typical top-level folder structure might include a **run script** (`run.py`, `main.py`, or `index.py`) used to start the application.

Namespace and Scope

Scope dictates what parts of the code can access other parts.

Analogy (House/City):

- The **city** is the whole system.
- A **class/function** is an individual **house**.
- Anything **outside** the house (e.g., a public park or public road) can be accessed by any program/house.
- Anything created **inside** one house (e.g., a specific function or variable) is accessible only by the people/code inside that house.
- If a program tries to access something inside someone else's house, that other house must explicitly give permission.

VI. Python Environment Management

Virtual Environments (venv)

A **virtual environment (venv)** is a mechanism for isolating applications.

Problem Solved by venv (The Story): If you install third-party dependencies (borrowed code) directly onto the main operating system Python, all applications rely on that central version. If that central dependency receives an update (e.g., changes from orange version to red version), applications dependent on the previous orange version will break.

Solution: Instead of installing dependencies globally, each application gets its own small, standalone version of Python and its own isolated dependencies (a "virtualized environment"). This allows one application to use the older orange version while another uses the newer red version, preventing version conflicts. **Always work in a virtual environment**; it is a must-have practice in the Python ecosystem.

Traditional Way to Create and Activate (venv):

1. **Create Folder:** Create a folder for the project (e.g., `01_virtual`).
2. **Open Terminal:** Open an integrated terminal in that folder.
3. **Creation Command:** Use the `venv` program, which comes automatically with Python. The new virtual environment folder is often called `venv` or `.venv`.

Command to create the virtual environment:

```
python3 -m venv .venv # or python -m venv .venv (on Windows)
```

4. **Activation:** This loads the virtual environment, showing `(venv)` in the terminal, indicating the system is using the isolated Python version.

Operating System	Activation Command (Assuming folder is `.venv`)	Citation
Windows	`.venv/Scripts/activate` (using forward slashes)	
Mac	`source .venv/bin/activate`	

5. **Deactivation:** To leave the virtual environment:

```
deactivate
```

The virtual environment folder (`.venv`) can be safely deleted; it does not need to be shipped with the software.

Managing Dependencies (requirements.txt)

Instead of installing modules one by one (e.g., `pip install flask`), developers use a standard method to ship dependencies.

1. **Create `requirements.txt`:** This is a simple text file (though the name is standard, it can technically be named anything, like `hitesh.txt`).
2. **List Dependencies:** Write the required modules inside, optionally specifying versions.

Example `requirements.txt`:

```
requests==2.31.0 # Specific version
flask==3.0.0
```

3. **Installation Command:** Use `pip install` with the `-r` flag to install modules from the requirements file.

```
pip install -r requirements.txt
```

This is how software is shipped: developers only ship the Python code and the `requirements.txt` file; the user creates a fresh virtual environment and installs dependencies from the file.

Modern Approach (UV)

The `venv` approach is traditional (used seven or eight years ago). A newer, more powerful tool called **UV** is gaining popularity and works much smoother, handling installation and environment setup without the manual steps described above.

VII. Python Style and Philosophy

PEP 8 (Style Guide)

PEP 8 is the style guideline for Python code, documenting the do's and don'ts of writing Python. This is typically studied *after* one is comfortable with Python, not by a beginner.

Key recommendations in PEP 8:

1. **Indentation:** Always use **four spaces**, never tabs.
2. **Naming:** Use **better meaningful names** for methods, functions, and classes (e.g., use `chai` instead of `C1`).
3. **Formatters:** Use formatters (like Black, rough, flakate) to automatically make code beautiful and compliant with PEP 8.

The Zen of Python

The **Zen of Python** is the "Pythonic way" of writing code. It can be seen by running `import this` in the Python shell.

The philosophy emphasizes simplicity and readability. Key tenets include:

- **Beautiful is better than ugly**.
- **Explicit is better than implicit**.
- **Simple is better than complex**.
- **Complex is better than complicated**.
- **Flat is better than nested**.

The main takeaway is to **always write simple code**. The **most beautiful code is the simplest code**; it should be readable so that anybody can understand what is happening.

VIII. Why Python? (Advantages)

Python is super easy to learn.

1. **Portability:** Once written, Python programs can run on a variety of operating systems, including **Windows, Mac, and Linux**. The only major difference in execution across operating systems is usually just the file path syntax (forward slashes in Windows vs. backslashes in Mac).
2. **Readability:** Python is very **readable**. If you can read English, you can definitely read Python code. The code is predictable.
3. **Productivity:** Python was invented to be **productive friendly**. It has less verbose code compared to languages like Java or C, and a decent level of Python can be picked up quickly (e.g., within a week).
4. **Standard Library (STL):** Python has a huge, rich, and extensive set of standard libraries and open-source code written by generous programmers. This code is available for commercial use. This availability of built-in tools is a major reason for its popularity in **data science, machine learning, and AI**.
5. **Multi-use/Flexibility:** Python is flexible and used for various applications, not just data science or terminal work. It can be used for **full-stack web apps, automation, data manipulation, CSVs, and**

machine learning.
6. **"Chai Level Happiness":** The instructor's most favorite reason is the joy and "Chai level happiness" that comes from writing code in the language.

3 Python Data Types

I. Core Python Concepts: Objects, Mutability, and Immutability

Understanding Objects in Python

In Python, the concept is that **everything is an object**. Every object must have a few defining properties:

1. **Identity (Unique ID):** Every single object has a unique ID. This identity helps to figure out whether an object is changeable or not changeable.
2. **Type:** Every object has a unique type (e.g., black tea, green tea, herbal tea, or numbers, strings).
3. **Value:** Each object will have some value (e.g., 2, hitesh, chai code). The value could be empty.

Mutability vs. Immutability

This is a fundamental concept in Python concerning how data is stored.

- **Mutable:** Changeable.
- **Immutable:** Not changeable.

A lot of people misunderstand mutability and immutability.

Rule for Validation: You must **always** validate mutability/immutability using the **identity (ID)**, **not** the value. Looking at the value is a wrong indicator and a common mistake for beginners.

- If the identity is the **same**, the value did not fundamentally change (it is mutable/changeable in memory).
- If the identity is **different**, a new object (a new reference) was created (it is immutable/not changeable).

Example: Immutable Numbers (Integers)

Numbers (integers) are considered **immutable**. The existence of the number in memory cannot be changed. When a variable assigned to a number is changed, Python creates a new number and changes the variable's reference to point to the new memory location.

Code Example of Immutable Change (Reference Change):

1. Initial variable definition:

   ```
   sugar_amount = 2
   ```

   (In memory, the value `2` is created, and `sugar_amount` points to it).
2. Changing the variable:

```
sugar_amount = 12
```

(Python creates a new number `12`. `sugar_amount` now points to `12`, but `2` remains unchanged and immutable in memory).

What seems to be changing in the world of immutable data types is the **reference**, not the actual value itself.

Tracking Identity (ID): To find the identity of an object, the `id()` keyword is used.

```
# Assuming this code is in chapter_one.py
sugar_amount = 2
# ... print initial sugar and ID
sugar_amount = 12
# ... print second sugar and ID

# Finding IDs
print(f"ID of 2: {id(2)}")
print(f"ID of 12: {id(12)}")
```

The IDs of 2 and 12 will be unique/different. The variable is pointing to a different memory location.

II. Python Numeric Data Types

Numbers are an important topic in programming.

Types of Numbers

1. **Integers:** Regular numbers, referred to by programmers with the fancy name "integers".
2. **Booleans:** True/False values.
3. **Real Numbers / Floating Point Numbers:** Numbers with decimals (used where precision is important, like stock prices or temperature).
4. **Complex Numbers:** Numbers with a real part and a fictitious/imaginary (iota) part (e.g., `2 + 3J`). These are rarely used and usually restricted to scientific/mathematical use cases.

Integer Operations

Basic arithmetic operations are available.

Code Examples of Integer Operations (in `chapter_three.py`):

Operation	Operator	Example Code	Concept
Addition	`+`	`total_grams = black_tea_grams + ginger_grams`	Adds two things.
Subtraction	`-`	`remaining_T = black_tea_grams - ginger_grams`	Standard subtraction.

Operation	Operator	Example Code	Concept
Multiplication	`*`	`*`	The sign of multiplication is the asterisk (`*`).
True Division	`/`	`milk_per_serving = milk_liter / servings`	Gives the exact decimal result (e.g., 1.75).
Floor Division	`//`	`bags_per_pot = total_teabag // pots`	Returns the result without caring about the decimal/remainder.
Modulo (Remainder)	`%`	`leftover_pods = total_cardamom_pods % pods_per_cup`	Gives the remainder of the division (the leftover).
Exponential (Power)	`**`	`powerful_flavor = base_flavor_strength ** scale_factor`	Calculates the power (e.g., 2 to the power of 3: 2 * 2 * 2).

Readability Improvement in Integers: Python allows separating large numbers using underscores (`_`) to improve readability. The number is still treated as without underscores.

```
total_tea_leaves_harvested = 1_000_000_000
# This improves readability.
```

Boolean Operations

Boolean values are `True` and `False`. The first letter is capitalized.

- `True` is represented as `1`.
- `False` is represented as `0`.

Adding a Boolean value to a number results in an **upcasting** where `True` becomes 1.

Code Example (Upcasting):

```
stir_count = 5
is_boiling = True

total_actions = stir_count + is_boiling # True is upcast to 1
# total_actions will be 6
```

Boolean Conversion (`bool()`): The `bool()` function (a method or function) can convert other values to Boolean.

- `0` and the keyword `None` are represented as `False`.
- Any non-zero number or non-empty string (e.g., `11`, `"Hitesh"`) is considered `True`.

Logical Operations (Boolean Algebra):

1. **AND:** Both conditions must be true (e.g., Tea AND Biscuit).
2. **OR:** Any one condition can be true (e.g., Tea OR Coffee).
3. **NOT:** Converts True to False, or False to True.

Code Example (AND operation):

```
water_hot = True
tea_added = False # Not ready to serve

can_serve = water_hot and tea_added
# can_serve will be False

tea_added = True
# Now can_serve is True
```

Floating Point Numbers (Real Numbers)

These require precision. When dealing with high precision numbers, standard subtraction may yield unexpected results due to how Python calculates precision.

Code Example (Precision):

```
ideal_temp = 95.5
current_temp = 95.49999999999999 # A lot of nines

difference_temperature = ideal_temp - current_temp
# The result might be unexpected (not exactly 0.0) due to precision.
```

For higher precision, developers must borrow code written by others (third-party packages or built-in modules).

Modules for Precision:

- **`sys`:** You can import `sys` and use `sys.float_info` to get system float information (this varies system to system).
- **`fractions`:** Used to deal with fractions, supporting high decimal points.

  ```
  from fractions import Fraction
  ```

- **`decimal`:** Powerful for dealing with huge numbers of decimal values.

  ```
  from decimal import Decimal
  ```

III. Python String Data Type

The string data type handles text. Strings are **immutable**; they cannot be changed, and operations always create a new reference in memory.

String Basics

Anything in double quotes (or single quotes, generally) is a string (e.g., `"ginger chai"`).

Indexing

Indexing means that **each letter in the string is represented by a number**.

- Indexing always starts from **zero** (0th letter).
- Square brackets (`[]`) are used for indexing.

Slicing

Slicing is used to grab parts of a string. The structure inside the square brackets is `[start:end:step]`.

Important Rule: The **last number (end index) is not inclusive** in Python indexing/slicing/range.

- **Start:** Where slicing begins (default is 0 if omitted).
- **End:** Where slicing stops (the character at the end index is excluded). If the end is omitted, it goes until the end of the string.
- **Step:** How many characters to move at a time (1 means every character, 2 means every second character).

Negative Indexing/Slicing (Reversing the String):

Using a negative step (e.g., `-1`) is a shorthand notation for reversing the whole string.

```
# Example: Reversing a string
chai_description = "aromatic and bold"
reversed_string = chai_description[::-1]
# Output: "dlob dna citamora"
```

String Encoding and Decoding

Encoding is necessary when dealing with **special characters or symbols** from different languages (like Spanish tildes, Hindi, Japanese, or Mandarin characters).

1. **Encode:** The `encode()` function converts the text into an encoded string (usually `UTF-8` or `UTF-16`).
2. **Decode:** To properly view or use the encoded text, it must be converted back using the `decode()` function with the same encoding.

Code Example (Encoding/Decoding):

```
label_text = "chai spécial" # Using a special character

# 1. Encoding
```

```
encoded_label = label_text.encode('utf8')
# The encoded label will look different, but guarantees character integrity.

# 2. Decoding
decoded_label = encoded_label.decode('utf8')
# This is the proper way to handle special symbols in a string.
```

IV. Python Tuple Data Type

Tuples are another data type used in Python.

Tuple Characteristics

- **Syntax:** Tuples are defined using **parentheses** (`()`).
- **Immutability:** Tuples are **immutable** (they cannot be changed).

Unpacking Tuples

Tuples allow easy extraction of values into multiple variables (unpacking). The number of variables must match the number of values in the tuple.

Code Example (Unpacking):

```
masala_spices = ("cardamom", "clove", "cinnamon")

spice_one, spice_two, spice_three = masala_spices

# Variables are now allocated: spice_one="cardamom", etc.
```

Variable Swapping

Tuples enable a unique and neat trick in Python: swapping variable values without needing a third temporary variable.

Code Example (Swapping):

```
ginger_ratio, cardamom_ratio = 2, 1

# Swap the values
ginger_ratio, cardamom_ratio = cardamom_ratio, ginger_ratio
# Now ginger_ratio is 1 and cardamom_ratio is 2
```

Membership Testing

You can check if a member exists within a tuple using the `in` keyword. This check is **case sensitive**.

Code Example (Membership Test):

```
# masala_spices = ("cardamom", "clove", "cinnamon")
is_present = "cinnamon" in masala_spices
# is_present is True

is_present_case_sensitive = "Cinnamon" in masala_spices
# is_present_case_sensitive is False
```

V. Python List Data Type (Mutable Sequence)

Lists are the first sequence type studied that is **mutable**. Lists can be changed after the memory reference is done, which is why many methods (like append, insert) exist for them.

- **Alternate Name:** Lists are often known as **arrays** in other languages, but in Python, they are called lists. They have the same data structure and definition.
- **Characteristics:** Lists can mix and match different data types. They can be reordered and changed.
- **Indexing:** Similar to strings, list elements are accessed via indexing, starting at position `0`.

List Methods

Method	Description	Example
`append()`	Adds an element to the **very end** of the list.	`ingredients.append("sugar")`
`remove()`	Finds and removes the specified element, regardless of its position.	`ingredients.remove("water")`
`extend()`	Combines one list with another existing list.	`chai_ingredients.extend(spice_options)`
`insert(index, element)`	Adds an element at a **specific index** (position). Elements to the right of the insertion point are shifted.	`chai_ingredients.insert(2, "black tea")`
`pop()`	Removes and **returns** the **last element** of the list.	`last_added = chai_ingredients.pop()`
`reverse()`	Reverses the order of the list elements **in place** (the list itself is changed and nothing is returned).	`chai_ingredients.reverse()`
`sort()`	Sorts the elements of the list (e.g., alphabetically).	`chai_ingredients.sort()`

Built-in Functions for Lists

Python has built-in functions that can operate on lists (arrays of numbers).

- `max(list)`: Returns the maximum value in the list.
- `min(list)`: Returns the minimum value in the list.

List Manipulation using Byte Arrays

It is possible to convert a string into a list-like structure of bytes using `bytearray()`. This is rarely used but exists.

- **Creation:** `bytearray()` uses parentheses (it is a method), and the string input must be prefixed with `B` for byte array.

  ```
  raw_spice_data = bytearray(B"cinnamon")
  ```

- **Characteristics:** The `bytearray` type is a **mutable sequence of integers**. It breaks the string down into characters.
- **Operations (Mutability):** Operations like `replace()` can be performed on a byte array. When a method runs on a byte array, it **returns a new array of bytes**. To see the change, the result must be reassigned back to the variable.

Code Example (Byte Array Manipulation):

```
raw_spice_data = bytearray(b"cinnamon")

# Operation returns a new value, so we must reassign it.
raw_spice_data = raw_spice_data.replace(b"cina", b"Cardamom")
# Printing raw_spice_data now shows the replacement.
```

VI. Python Set Data Type

Sets are a data type that represents a collection of things.

Set Characteristics

- **Uniqueness:** The most defining characteristic of a set is that **everything is unique**.
- **Mutability:** Sets are **mutable** (changeable). The identity remains the same even after elements are added. You can constantly add or mutate the elements in memory.
- **Syntax:** Sets are defined using curly braces (`{}`) or the `set()` constructor.
- **Order:** Sets are an unordered collection.

Set Operations (Mathematical Concepts)

Set operations are based on mathematical concepts.

Operation	Concept	Operator	Example Code
Union	Combines everything from both sets. If an element is common, it is not repeated (maintains uniqueness).	Pipe (`|`)	`all_spices_box = essential_spices | optional_spices`

Operation	Concept	Operator	Example Code
Intersection	Returns only the elements that are common to both sets.	Ampersand (`&`)	`common_spices = essential_spice & optional_spices`
Difference	Returns elements present in the first set but **not** in the second set (removes common elements).	Minus sign (`-`)	`only_in_essential = essential_spices - optional_spices`

Frozen Set

A `frozenset` is an available concept that represents an **immutable, unordered collection of unique elements**. It is used when you want to freeze a set.

Membership Testing

Membership testing works the same way as with tuples, using the `in` keyword.

VII. Python Dictionary Data Type

Dictionaries provide **named-based indexing** instead of numerical indexing (0, 1, 2...). They store data where each piece is pointed to by a name.

- **Syntax:** Dictionaries are created using curly braces (`{}`) or the `dict()` function.
- **Order:** Order generally does not matter in a dictionary because elements are referenced by name (key).

Structure and Creation

Data is stored in **key-value pairs**.

- **Key:** The name used to reference the data (e.g., `type`, `size`, `sugar`).
- **Value:** The data stored (e.g., `masala chai`, `large`, `two`).

Code Example (Dictionary Creation):

```
chai_order = dict(type="masala chai", size="large", sugar=2)

# Or using curly braces (empty dictionary)
chai_recipe = {}
```

Accessing, Adding, and Deleting Data

1. **Adding Data:** Use square brackets (`[]`) with the key name to define the value.

   ```
   chai_recipe["base"] = "black tea"
   ```

2. **Accessing Data:** Use square brackets (`[]`) with the key name.

```
recipe_base = chai_recipe["base"] # Gets "black tea"
```

3. **Deleting Data:** Use the `del` keyword followed by the dictionary name and the key in square brackets.

```
del chai_recipe["liquid"]
```

Dictionary Methods for Information Retrieval

Method	Description	Output Format
`keys()`	Returns a list-like collection of all the keys in the dictionary.	`DICT_keys` (A list/array containing the keys).
`values()`	Returns a list-like collection of all the values in the dictionary.	`DICT_values` (A list/array containing the values).
`items()`	Returns all key-value pairs.	A whole list outside, containing tuples of key-value pairs inside.
`popitem()`	Removes and returns the **last item** (key-value pair) from the dictionary.	Returns the removed item.
`update()`	Updates the dictionary with key-value pairs from another source (e.g., another dictionary).	Merges the new data; existing keys are overwritten, and new keys are added.

Safe Value Retrieval using `get()`:

If you try to access a key that doesn't exist using square bracket indexing (`chai_order["non_existent_key"]`), the application will crash.

The **safe way** is to use the `get()` method. This method allows you to specify a default value to return if the key is not found, preventing crashes.

Code Example (Safe Retrieval):

```
# Assuming 'customer_note' does not exist in chai_order
customer_note = chai_order.get("customer_note", "No note was given")
# If found, it returns the value; otherwise, it returns "No note was given".
```

Membership Testing

Membership testing (checking if a key exists) can be performed using the `in` keyword.

Code Example:

```
is_sugar_present = "sugar" in chai_order
# is_sugar_present is True
```

VIII. Advanced Data Types and Modules

Advanced data types in Python often require bringing in code written by somebody else, known as **third-party modules**. A module is just a fancy name for someone else's code brought into the program.

These advanced topics are usually encountered after having six months of Python experience.

Time and Date Modules

These modules manipulate date, time, duration, and calendars.

Advanced Data Type/Package	Description	Concept/Utility
date time	Includes both date and time at the same time.	Manipulate date time.
time	Separately handles time.	Manipulate time.
calendar	A data type for calendar manipulation.	Manipulate calendar.
time delta	A package that works with time duration (deltas simply mean difference between two things).	Used to discuss duration (e.g., order fetch time vs. delivery time, or program run time).
arrow	A utility package that can be imported.	Provides UTC time and conversion between time zones.
date util	Another utility package that can be imported.	Works similarly to `arrow`.

Using the `arrow` Module (Example):

To use modules, an `import` statement is needed.

```
import arrow

brewing_time = arrow.utcnow()
# brewing_time now holds the UTC time.

# Converting to another time zone (e.g., Europe/Rome)
converted_time = brewing_time.to('Europe/Rome')
```

Collections Module

`collections` is a major category of advanced data types. These data types are built on top of default Python types (like strings and lists).

Available Data Types in `collections` (from official documentation):

- `namedTuple`
- `deck` (called DEC, not DQ)
- `ChainMap`
- `Counter`
- `OrderedDict`
- `defaultdict`
- `UserDict` (and many more)

Using `namedtuple` (Example):

The import statement usually goes at the top of the code.

```
from collections import namedtuple

# Define the structure of the named tuple:
# Arguments are: Name of the tuple (as a string), and an iterable (list/array) of field names.
ChaiProfile = namedtuple('Chai Profile', ['flavor', 'aroma', 'color'])

# This provides an additional data type.
```

IX. Operator Overloading

Operator overloading occurs when an operator (like `+` or `*`) is used for **doing more than one task**.

Addition (`+`) Overloading on Lists

The plus operator (`+`) is designed to add numbers. When used on lists, it performs concatenation (combining the lists).

Code Example (Concatenation):

```
base_liquid = ["water", "milk"]
extra_flavor = ["ginger"]

# Operator overloading: using '+' to combine lists
liquid_mix = base_liquid + extra_flavor
# Result: ['water', 'milk', 'ginger']
```

Multiplication (`*`) Overloading on Lists

The multiplication operator (`*`) can be used on a list and a number. This results in the list being multiplied (repeated) that many times. The original order of elements within the list is maintained for each repetition.

Code Example (Repetition/Multiplication):

```python
strong_tea = ["black tea", "water"]

# Operator overloading: using '*' to multiply the list
strong_brew = strong_tea * 3
# Result: ['black tea', 'water', 'black tea', 'water', 'black tea', 'water']
```

The `operator` Module

More operations can be brought in using the `operator` module. For instance, `itemgetter` can be imported.

```python
from operator import itemgetter
```

The documentation describes `itemgetter` as returning "a callable object that fetches the given item from its operand". This utility is often seen in source codes and is used to sort based on a variety of steps.

Prompts

```
Create super depth notes in Markdown (.md) format with 100% information preserved,
no loss. Use simple grammar and keep everything clear, direct, and well-
structured. using headings, subheadings,paragraphs, statements and code blocks
when needed. Include every detail, definition, example, and step exactly from the
source. transform the given content into clean, readable .md format.
```